SPARSKIT CCA Component *

Jason Jones† Masha Sosonkina †

1 Introduction

SPARSKIT[6], developed by Yousef Saad at the University of Minnesota, is a basic toolkit for sequential sparse matrix computations and is widely used in scientific community. Written in Fortran 77 and having a cumbersome interface, it is considered, however, a legacy code. Our objective is to enable its wider usage in modern applications and to facilitate further development of SPARSKIT. Tools available from the Common Component Architecture (CCA) [2] such as Babel [4], Chasm [5], and Ccafe [2] framework provide a means of creating a SPARSKIT package that can be used by many different programming languages. Taking SPARSKIT and applying an object-oriented style of design, we have created a package that can be accessed by many different object-oriented programming languages. There is also the possibility of writing extensions in these languages.

Related work. *Hypre* [3] and PETSc [1] are two software packages that also contain a suite of sparse matrix computation routines, including parallel implementations. Their original design uses modern software engineering tools and practices, and thus yields to component design easier than a legacy code does so. Fundamental questions, however, – such as component structure, interaction, argument lists – require a general solution that would satisfy the needs of a majority of applications. For example, in the *Hypre* CCA interface, parameters are explicitly associated into name-value pairs before a call to a component. PETSc CCA interfaces have not been made public yet. In this paper, we propose another possible interface to sparse matrix computations and put forward some issues that we face in our design process.

2 SPARSKIT Component Design

To create a SPARSKIT component, a re-design of the basic structure was necessary. Previously, the directory and code structure were that of a non-object-oriented software package due to Fortran77 being non-object oriented. We created a Java-like structure such that all the members of the same "class" perform a particular type of task in SPARSKIT. For example, preconditioning, iterative accelerating, or matrix formatting – each belongs to its own "class", i.e., implements a particular interface. The former two are used to solve a sparse linear system *approximately*, by performing a series of iterations, with preconditioning serving as a transformation of the linear system that may yield fewer iterations.

*This work was supported in part by the U.S. Department of Energy under Contract W-7405-ENG-82

†Ames Laboratory, Iowa State University, Ames, IA 50011, {jonesj,masha}@scl.ameslab.gov.

This design allows for more functionality, possibly developed in another language, to be supported by the CCA tools and be added transparently to the user.

In addition, the user may be able to take advantage of the new functionality without changing the source code due to a standard set of arguments and naming conventions, which we have developed for each interface. This issue is very important for easy and uniform access to SPARSKIT routines. Since the preconditioners in original SPARSKIT accept varying number and types of arguments, it was much harder to design a uniform interface for preconditioners than for accelerators, which typically accept the same arguments and all use reverse communication (see [6] for more details). Our implementation of SPARSKIT interfaces also simplifies SPARSKIT usage by removing the need for temporary work arrays, which are often passed as subroutine arguments in Fortran77. In a nutshell, we propose the following argument list and names for a general preconditioner interface (also called Port in the CCA terminology): prefixes are the names as appear in SPARSKIT and arguments consist of the input matrix represented as three arrays in a sparse storage format, an array of integer parameters, an array of double-valued parameters, and the output matrix, also represented as three arrays. We use sparse matrix representation as a set of arrays, similar to the one in Fortran77, to avoid the potential need for copying between different representations. The following example shows how this port design can be accessed in C++:

```
skit::components::ilut ilut = skit::components::ilut::_create();
ilut.create(a,ja,ia,lfil,droptol,alu,jlu,ju);
```

The same example will look like this when accessed using Python:

```
import skit.components.ilut
...
ilut = skit.componenents.ilut.ilut()
ilut.create(a,ja,ia,lfil,droptol,alu,jlu,ju);
```

In this code fragment, `ilut` stands for the incomplete LU factorization with dual threshold (ILUT) [7] and is one of the SPARSKIT preconditioners [6]. Using our component in other object-oriented languages has similar syntax as does using the other parts of our component.

One feature of the CCA tools is that they use a standard type for arrays across all the supported languages. These arrays are called SIDL arrays and they allow for easy interoperability on arrays between the supported languages and a standard way of handling those arrays in each language. For SPARSKIT to be implemented as a CCA component, all the Fortran77 arrays had to be converted into SIDL arrays. We also opted to shift the variables that were used to index the arrays to ensure that they were indexing the correct element in the arrays. Converting the Fortran77 arrays to SIDL arrays can be somewhat cumbersome and must be done manually at this point. We are currently investigating ways to help automate this process to speed up the process of componentizing current SPARSKIT and possible future additions to the package.

Table 1: Execution Times on Generated Matrix

nnz	SKIT, sec	SKIT-CCA, sec	diff
41440	0.027	0.04	0.013
50257	0.035	0.05	0.015
65241	0.044	0.06	0.016
95205	0.083	0.12	0.037
132660	0.14	0.19	0.050

3 Test Results

We have successfully created a CCA Port prototype of a SPARSKIT component that implements multiple accelerators, preconditioners, format conversion, and matrix-vector multiplication components. Clients have also been written in C, C++, and Python to demonstrate the interoperability achieved by our SPARSKIT component. The numbers in Table 1 show the results of tests ran under the original SPARSKIT compared with the same tests run under our SPARSKIT Component (column `SKIT`) using a C++ client (column `SKIT-CCA`). The tests were run on a Pentium 4 3.2GHz with 1 GB of RAM running Debian Sarge/Testing with the 2.6.8 Linux kernel. The CCA-Tools 0.5.6 rc4 sumo tarball was used as the base and the tests were run using matrices generated by SPARSKIT `gen57pt` function (column *nnz* in Table 1 refers to the number of non-zero elements in the matrix), ILUT as the preconditioner, and FGMRES [7] as the accelerator. The execution times are from an average of 5 runs on each array size. The overhead incurred by the Component framework appears to increase steadily as the size of the arrays tested increase. This can be seen in the difference between the running time for SKIT and SKIT-CCA (see column `diff` in Table 1).

4 Open Issues

In the future we plan to use the Chasm Tools to aid in porting the rest of SPARSKIT into our component structure. At this time, however, Chasm does not have full Fortran77 support. We are currently communicating with Craig Rasmussen from the Chasm team in efforts to address this issue. We are also investigating the applicability and potential of a more complex interface structure, interface hierarchy. In particular, we want to have "external" and "internal" CCA ports. External ports would be exposed to all other CCA ports in the framework and internal ports would be local to the package they are used in. Internal ports can be used to give a default implementation of a specific port if no other components providing that port are available. We envision internal and external ports being similar to how `public` and `private` are represented in object-oriented languages. The idea of these two types of Ports is motivated by the recursive nature of some preconditioners to be added to SPARSKIT. In particular, Algebraic Recursive Multilevel Solver (ARMS) [8] has a multilevel structure, each level of which may be defined recursively by a wide range of sparse or dense linear system solution routines.

References

[1] S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L. McInnes, B. Smith, and H. Zhang. PETSc web page. `http://www.mcs.anl.gov/petsc`.

[2] D. Bernholdt, W. Elwasif, J. Kohl, and T. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, 2002.

[3] E. Chow, A. Cleary, and R. Falgout. *hypre* User's manual, version 1.6.0. Technical Report UCRL-MA-137155, Lawrence Livermore National Laboratory, Livermore, CA, 1998.

[4] T. Dahlgren, T. Epperly, G. Kumfert, , and L. Leek. Babel users' guide. `http://www.llnl.gov/CASC/components/docs/users\_guide/index.html`.

[5] C. Rasmussen and M. Sottile. Chasm reference manual. `http://chasm-interop.sourceforge.net/F90ArrayDesc_API.pdf`.

[6] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps`.

[7] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[8] Y. Saad and B. Suchomel. ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Numerical linear algebra with applications*, 9(5):359–378, July/August 2002.